EnosStack: A LAMP-like stack for the experimenter

Ronan-Alexandre Cherrueau, Matthieu Simonin, Alexandre van Kempen

EnosStack: A LAMP-like stack for the experimenter

Ronan-Alexandre Cherrueau
Inria, Mines Nantes, LINA
Nantes, France
ronan-alexandre.cherrueau@inria.fr

Matthieu Simonin
Univ Rennes, Inria, CNRS, IRISA
Rennes, France
matthieu.simonin@inria.fr

Alexandre van Kempen
Inria, Mines Nantes, LINA
Nantes, France
alexandre.van-kempen@inria.fr

***Abstract*—Reproducibility and repeatability dramatically increase the value of scientific experiments, but remain two challenging goals for the experimenters. Similar to the LAMP stack that considerably eased the web developers life, in this paper, we advocate the need of an analogous software stack to help the experimenters making reproducible research. We propose the EnosStack, an open source software stack especially designed for reproducible scientific experiments. EnosStack enables to easily describe experimental workflows meant to be re-used, while abstracting the underlying infrastructure running them. Being able to switch experiments from a local to a real testbed deployment greatly lower code development and validation time. We describe the abstractions that have driven its design, before presenting a real experiment we deployed on Grid'5000 to illustrate its usefulness. We also provide all the experiment code, data and results to the community.**

***Index Terms*—Repeatability, Reproducibility, Application deployment, Performance, Grid5000, Chameleon**

I. Introduction

In early 2000, the LAMP stack was considered as the Graal for web developers. The stack was composed of Linux as Operating System, Apache as Web server, MySQL as database backend and PHP as scripting language. The stack was packaged all together in the major linux distributions increasing its adoption. It was opinionated but decoupled clearly the different parts needed to build a web application. The application logic later moved to dedicated frameworks (e.g Django, Ruby On Rails) which further abstract the database backend through the use of ORMs (Object-Relationnal Mapping) As a consequence the efficiency of the developer increased due to easy switching between a local development mode and production deployment.

In this paper, we argue that experimenters also require their LAMP stack to assist them in making reproducible research. Indeed, while reproducible research is a key component of the scientific method, it is far from being the norm, especially in the context of computational science [**?**], [**?**]. Even if experimenters are committed to automate their experiment code, some tasks remain tedious [**?**]. First, maintaining a code to keep up with the development of the upstream software code can be time consuming (e.g OpenStack is released every six months). Second, sharing the code with others requires to follow coding and packaging gold standards [**?**]. The latter helps experimenting in a repeatable way, the former addresses the problem of revisiting results in different time frames.

Leveraging concepts introduced in frameworks such as OMF [**?**], we propose the EnosStack, an open source software stack, as a solution to such challenges by following the model of the LAMP stack. Parallely the authors of *Popper* [**?**] paved the way towards a set of common conventions that would ensure an experiment to be easily re-executed and validated. EnosStack is a pragmatic framework in which such conventions can be implemented.

The software stack is composed of a group of open source software that are typically installed together to ease the orchestration of reproducible experiments. It includes Python, Ansible, Docker and the *EnosLib*, a library we specially developed for this work. Large distributed applications were the initial targets of the EnosStack, but it appears flexible enough to target all kind of applications.

In order to introduce the concepts of EnosStack while easing the description of its implementation, we present a red-thread example that we will rely on throughout the paper to illustrate more concretely various concepts. This example consists in evaluating the scalability of a message-oriented-middleware (MOM, e.g. RabbitMQ[1]) stressed by multiple agents communicating through it. To perform this experiment on a real testbed (e.g Chameleon [**?**], Grid'5000 [**?**]), an experimenter would need to configure all the agents as well as the MOM and then deploy some code that benchmarks it for a given number of agents. This process would have to be repeated with an increasing number of agents until reaching the limit of the MOM. During these experiments, various metrics would have to be monitored and backed up before the resulting data analysis. As shown in the next sections, a such seemingly simple experiment leads to numerous design challenges when having to be deployed on real testbeds.

More specifically, the contributions of this paper are as follows:

- We identify key concepts and good practices to deploy experimental workflows.
- We introduce the EnosStack, a novel LAMP-like stack for the experimenter.
- We present a use case experiment that evaluates the scalability of a RabbitMQ bus, deployed on Grid'5000 thanks to the EnosStack.

The remainder of the paper is structured as follows: section II discusses high level requirements, section III introduce concrete technologies for EnosStack. The section IV presents the most important concepts of the EnosStack that help in

[1] https://www.rabbitmq.com/

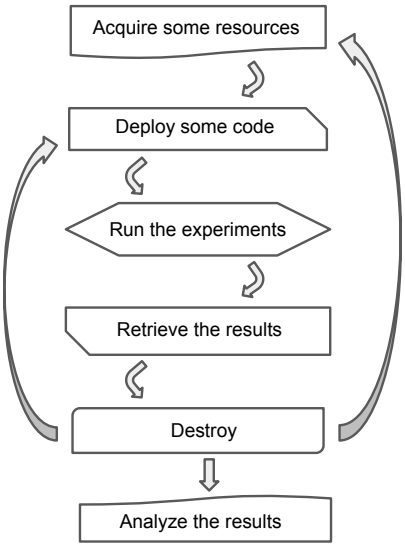

Fig. 1. Experimental workflow

fulfilling the previous requirements. Finally, section V will present the evaluation of the real deployment of our red thread example on Grid'5000 before concluding in Section VI.

II. High-level requirements

We envision the process of creating an experiment in four steps: Designing, validating, scaling and sharing. Each step is in general challenging [**?**] thus a stack-for-experimenter must ease each one of them. In this section, we outline some high-level requirements that emerge from the above steps.

(i) Designing the experiment: A typical experiment can be expressed as a workflow of tasks as in Fig. 1. This workflow starts by (1) getting some resources from a specific testbed, (2) deploying the application itself and all the third-party software needed (e.g instrumentation), (3) running the wanted workload (4) retrieving the results, (5) destroy everything before iterating on all the previous steps for other configurations, or parameter set. Once the required parameter space of the experiment has been explored (6) the results analysis can finally be performed. The enactment of the above workflow must be fully automated and repeatable. As a result, being able to easily describe the workflow and to execute it without sacrificing fast iterations on the code is a strong requirement for the EnosStack.

(ii) Validating the experiment: The typical experimenting process can be split in two phases: first, the development phase and then, the production phase. In the development phase an experimenter needs to iterate over the code of the experiment until reaching the desired level of automation and correctness. In the production phase, the experimenter will actually run the code of the experimentation on the targeted platform (e.g Grid'5000, Chameleon). Most probably, experimenters will need to go back and forth between the two phases while refining their experimentation code (bug fixing, scalability issue, etc.). Production platforms are often shared between users and rarely offer a friendly development environment. Thus, skipping the first phase leads an experimenter to waste precious resources on the production platform and be less efficient while iterating on the code. As a consequence, being able to transparently switch between environments is another strong requirement for the EnosStack.

(iii) Scaling the experiment: After validating the workflow and its correct execution, experimenters likely need to scale it. This is typically achieved by deploying more processes of the application under study. To run those extra processes one can choose (a) to claim more compute resources to the underlying infrastructure to run them or (b) to increase the density of the processes on the servers. In both cases, an experimenter need a simple way to specify it and an effortless way to enforce it.

(iv) Sharing the experiment: By sharing, we mean sharing the whole experiment or part of it. The former provides reusability. It is catalyzed, for instance, when an easy path to install (e.g package) is offered or a friendly user experience (e.g command line interface) is provided. The latter refers to the modularity of a software project. In any cases, following any high-level language guidelines of software development leads to these properties. As a consequence, an experimenter who writes experiment as code needs to be given the same development experience as in a regular software project.

III. The EnosStack layers

The EnosStack's cornerstone is provided by the EnosLib, a library we specially developed for this work. Leveraging the EnosLib, we built the EnosStack by including well known open source software namely Python, Ansible and Docker. We describe hereafter these four components in more details.

1) Python: Python is a versatile language, that comes with one of the most mature package libraries. It is particularly adapted to scientific experiments thanks to powerful libraries tailored for data science such as numPy and sciPy [**?**]. Moreover, it neatly interfaces with Ansible thus making it the language of choice for the EnosStack.

2) Ansible: Ansible is an orchestration tool that automates software provisioning, application deployment and configuration management. We detail hereafter some of its concepts we will use throughout the paper. In Ansible, a *task* is an action that is remotely controlled on a machine. Tasks are launched on groups of nodes using a fork-join model. The sequence of tasks to run is described in a *playbook*. Ultimately, playbooks can be packaged alongside their required files and variables to form an Ansible *role*. Finally, an *Inventory* file allocate hosts into different groups to logically map physical nodes to Ansible roles.

Including Ansible in the EnosStack has been driven by the fact that, contrary to other similar tools, Ansible only requires an SSH connection as it follows a push model and is thus agentless. This avoids the need of any software installation on the targeted machines, making it very lightweight to setup. Additionally it's a robust software backed by a large community of users. Its role mechanism is particularly interesting as it provides a good level of modularity in the code produced or re-used from a third-party user.

3) Docker: Docker is an open-source project designed to ease the deployment and execution of applications by using containers. Containers are like packages that include all the libraries and other dependencies required by a given application. Being self-sufficient, containers offer an inherent portability, while considerably easing the lifecycle of containerized applications. These are properties particularly suitable in our context of reproducible experiments as explained in [**?**]. Moreover they provide a unified abstraction for the life-cycle management of programs. This eases the description of the deployment of multi-containers applications.

4) Enoslib: The Enoslib [**?**] is an open source library we developed that enables a developer to describe and run an experimentation workflow on different infrastructures. It abstracts the underlying infrastructure to a developer, such that only few changes are required to migrate an experiment from one testbed to another testbed. It also allows the experimenter to define tasks and combined them to form the desired experimental workflow in a way that allows incremental code development. Enoslib takes its roots in EnOS [**?**] but goes beyond as it gets agnostic to the application under study.

IV. EnosStack abstractions

We now present the main abstractions the EnosStack rely on. In order to fulfill the requirements explained in the previous section, we carefully designed the EnosStack to abstract concepts such as *resources*, *services* or *tasks* as explained hereafter.

A. Resource Model

In the EnosStack, the concept of providers is the key that allows experimenter to switch from one execution context to another. EnosStack is shipped with three providers: Vagrant, Grid'5000, OpenStack. The former is mainly used in the development phase while the others will be preferably used in the production phase. Note that an implementation for the Chameleon platform is also possible and is inherited from the OpenStack provider. EnosStack models two types of resources as described in the next sections.

a) Machines: In EnosStack, machines are compute resources and can be grouped. A group is given a *count* and a set of *roles*. In the current implementation, machines in one group share the same hardware characteristics. A group is an abstract description and it is the provider responsability to concretize this description by : (1) claiming *count* machines from the underlying infrastructure (2) distributing those machines in the roles. The roles of the machines will be used at deployment time to differentiate the configuration to apply on each machine.

b) Networks: In EnosStack, networks provide connectivity between the different machines. Networks are assigned to machines, allowing to model non trivial topologies. Depending on the underlying provider used, the implementation may differ : bridge networks on Vagrant, vlans on Grid'5000 or private networks on OpenStack. Similar to a group of machines, a network can be given a set of roles. For example, these roles can be used at the deployment time to segregate the network traffic.

```
machines:
  - flavor: large
    count: 3
    roles:
      - mom,
      - telegraf
      - influxdb
    networks:
      - control_network
      - internal_network
  - flavor: tiny
    count: 10
    roles:
      - mom-agents
      - telegraf
    networks:
      - control_network
```

```
machines:
  - flavor: parasilo
    count: 3
    roles:
      - mom
      - telegraf
      - influxdb
    networks: [cn, in]
  - flavor: paravance
    count: 10
    roles:
      - mom-agents
      - telegraf
    networks: [cn]
networks:
  - id: cn
    roles:
      - control_network
    type: prod
  - id: in
    roles:
      - internal_network
    type: kavlan
```

Fig. 2. Resource descriptions for Vagrant (left) and Grid'5000 (right)

c) Example: Fig. 2 gives the declaration of the resources for the same experiment. Fig. 2(a) will start virtual machines and get two networks from the local VirtualBox hypervisor. Fig. 2(b) will deploy bare-metal servers on the Grid'5000 platform and get one isolated VLAN network (the second network is the default provided by Grid'5000). The description of machines is slightly different : size refers to a predefined amount of vCPU and Memory in the Vagrant case whereas cluster name is used in the Grid'5000 case. Also, network description for Grid'5000 allows to specify inhomogeneous network card configuration. More concretely here, specifying a *count* greater than one deploys several agents of the MOM in a clustered mode where the internal traffic (e.g replication traffic between RabbitMQ brokers) is isolated in a VLAN. Note also that the experimenter can give the two roles to the same network and traffic will not be isolated. This flexibility is allowed by the service abstraction presented in the next part.

B. Service model

During the deployment of an experiment, software will be installed and configured. Those software can be libraries but also running daemon processes that can optionally require a persistent storage to keep their states. In EnosStack, we call *service* an abstraction of one or several software that serves the same purpose regarding the application to deploy. A service has inputs (e.g configurations options for the encapsulated programs) and actions that transition the state of the service. This service abstraction is beneficial for both the extensibility of an experiment (e.g considering new scenarios) and re-usability (e.g using some building blocks in another experimental workflow).

Going back to the red thread example, a monitoring infrastructure is required to collect performance metrics. Since it's likely to be re-used in another context we model this

infrastructure as a service. It is composed of monitoring agents (Telegraf[2]), metrics collector (InfluxDB[3]) and a graphical frontend (Grafana[4]). Note that the monitoring service can be seen as a composite service composed of three sub-services: telegraf, influxdb and grafana. To ease the management of their life-cycle, we chose to containerize all the corresponding components. We also consider the following actions and semantics: *deploy* installs and configures all the programs according to the inputs, *backup* fetches the persistent storage of the metrics collector (the docker volume attached to the collector container) to keep all the metrics generated. Finally, *destroy* removes all the corresponding containers as well as their associated volumes.

At deployment time, EnosStack allows to map a service to a role and thus to all the machines in that role. More concretely, according to the description given in Fig. 2, the telegraf sub-service will be deployed on thirteen machines. At this point increasing the number of machines associated with the telegraf role will scale automatically the number of running monitoring agents. Additionally changing the monitoring agent software to an equivalent one is as simple as switching the monitoring agent service implementation. In conclusion, following the above service model allows to benefit from a certain degree of modularity and scalability.

C. Task model

A typical experiment follows a workflow of tasks as outlined in the section II. For instance, the experimenter of the red-thread example first claims resources on a testbed. Then, she deploys the MOM and monitoring services. Next, she stresses the MOM with a given number of agents, and backups results of the experiment. Finally, she destroys the experiment to start a new one in a fresh environment. The new experiment will follow a nearly similar workflow of tasks (only the stress task will vary by increasing the number of agents) which emphasis the need for *fast iterations* on a task to adapt it to new experiments.

In the EnosStack, a task is a python function. By itself, a python function already provides mechanisms for task's adaptation. This is the case of function parameters that adapt the task when they are given a value. But, such mechanism are not sufficient to provide fast iteration on an experiment. To get fast iteration, the experimenter needs facilities to easily stop, then adapt, and finally restart the experiment at the moment of a specific task. Indeed, during the development phase, an experimenter often has to exit the experiment execution, *e.g.*, to fix it, before returning to it. Unfortunately, this is not possible in a normal python program. If the experimenter stop the experimentation at a certain task, then she looses the execution state of that task (hidden by the python runtime environment) and cannot return directly to it latter. To get back to that task, she has to rerun the experimentation from the beginning. A rerun that is time consuming when the experimentation has heavy tasks such as deploying a specific Operating System on the testbed resources.

The EnosStack offers facilites to reifies the state of an experiment so that every task of the workflow is filled by a data structure that represents the execution state of that experiment. Executing a task modifies that data structure, that is then passed to the next task. The created data structure thus can be accessed by the experimenter, instead of being hidden in the python runtime environment. Thanks to that mechanism, an experimenter can easily recall a stopped task with its execution state, and thus restart the experimentation where she left it. For instance, the red-thread example exposes five tasks (see II). The first two one are in charge of getting resources from a testbed and deploying the application. When EnosStack executes the first task, it fills it with the resource model (see Fig. 2) and gets in return the concrete resources list. EnosStack then passes that resources list to the second task and gets in return, *e.g.*, the IP addresses of deployed services. EnosStack stores the data structure in the filesystem and thus, make it accessible to the experimenter. So, if the experimenter hits `ctrl+c` in the middle of the second task, she can then recall that second task with its execution state (*i.e.*, the concrete resources list) stored on the filesystem. Doing so proceeds the experimentation and save the need to restart the experimentation from the beginning. Also note that an experimenter can easily extend the EnosStack with other programming languages (*e.g.*, C, Go, Ruby) by accessing and modifying the reified state.

To fast iterate on experiments, the EnosStack also favors *idempotent* tasks. In computing, an operation is idempotent if it can be applied many times without changing its result. For instance, calling twice the deployment task of the MOM service will do nothing the second time, since the MOM has been already deployed the first time. The result is a deterministic run, even when tasks imply side effects. Getting a deterministic run helps during the development phase of an experimentation. It lets the experimenter develop interactively on its experimentation until it works as expected. This, once again, speeds up the development phase.

Facilities provided by the EnosStack are idempotent. For instance, providers, that get testbed resources (see IV-A), are idempotent. Thus, calling a provider twice will do nothing the second time. More generally, the EnosStack relies on Ansible modules to favor idempotent tasks. An ansible module is an idempotent action that can be performed locally or remotely (*e.g.*, copy a configuration file, start a container, execute a specific process, etc.). All together, Ansible modules form a Domain Specific Language (DSL) for scripting tasks that are idempotent by design.

V. Message Oriented Middleware evaluation

Benchmarking a software component is a common experiment that can provide a deeper understanding about the component behavior, its scalability, its reliability, etc. For this example, we chose to evaluate the scalability of a message-oriented-middleware, namely RabbitMQ, to illustrate how a

[2]https://www.influxdata.com/time-series-platform/telegraf/
[3]https://www.influxdata.com/time-series-platform/influxdb/
[4]https://grafana.com/

real experiment could be deployed thanks to the EnosStack. We describe this experiment in more details in the next section before presenting a related demo scenario and sharing the experiment material.

A. Experimentation

In order to illustrate the usefulness of the EnosStack, we present in this section some experiments we conducted in the context of the OpenStack performance working group. The experiments presented hereafter are part of a much broader test plan[5] that aims to evaluate the OpenStack message bus in the context of a massively distributed cloud architecture. We first detail some background about these experiments, and explain how we implemented them before presenting briefly some results.

1) Context: OpenStack is a cloud operating system, composed of many services that has to communicate between them. One way these services can communicate is through a Remote Procedure Call (RPC) system, built on top of a message bus. The OpenStack services relies on *Olso Messaging*, a library that provides a messaging API which supports RPC over a number of different messaging transports. In other words, Olso Messaging abstracts the underlying messaging middleware technology, allowing various messaging buses such as RabbitMQ, Qpid or ZeroMQ to be deployed. Being able to compare the performance of these different buses for different configurations would thus be highly valuable for the community. However, deploying such experiments in a repeatable way and at scale is a tedious task and can quickly become a time-consuming endeavor.

2) Implementation: We only describe here the experiments and results related to RabbitMQ as it is one of the most popular open source message bus. In order to stress the RabbitMQ bus, clients periodically sent RPC request messages to servers over the queue. To generate the RPC requests, we relied on *ombt* (Oslo Messaging Benchmarking Tool), an open source tool able to create RPC agents (clients or servers), and measure the latency and throughput of RPC transactions for each agent. In addition to these two metrics, we also collected numerous others, such as the CPU, memory, I/Os, etc. for every agent. Those metrics are collected using the monitoring service described in section IV-B. More widely, all the services were deployed in docker containers to ease the experiment management. Indeed, backuping the experiment consists in retrieving all the metrics and all the statistics from the RPC agents. This allows very deep post-mortem exploration or even in real-time. Destroying the experiment consists in removing all the running docker container as well as their attached states. Practically this means that the workflow (Fig. 1) can be repeated without the need to start fresh resources each time. This is particularly interesting in our cases because iterating over more than 50 sets of parameters were required. Thanks to the EnosStack, we were able to orchestrate in a fully automated way different executions of the experiment.

[5]https://docs.openstack.org/performance-docs/latest/test_plans/massively_distribute_rpc/plan.html

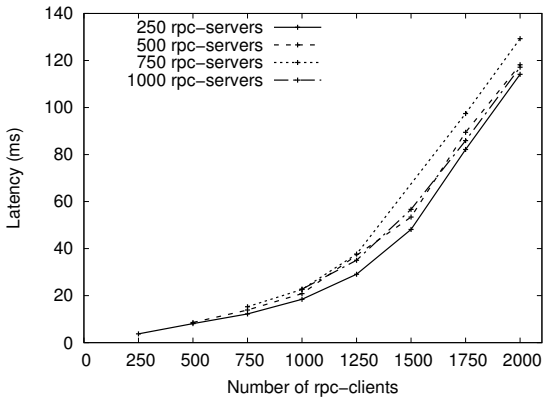

Fig. 3. Average Latency

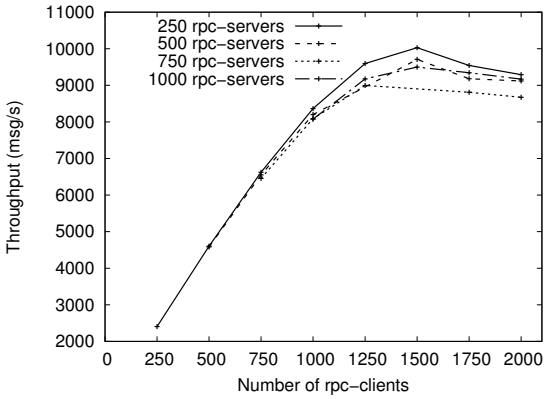

Fig. 4. Average Throughput

3) Results: We deployed these experiments on the Grid'5000 testbed on a cluster of 72 nodes, equipped with Intel Xeon E5 2.4Ghz, 128GB of memory and 10Gb connections. We emphasize that, leveraging the concept of providers in the EnOSLib, we first validated this workflow locally on Vagrant, before effectively deploying it on Grid'5000 on thousands of agents. This enabled us to greatly reduce the development time, while significantly easing the debugging process. We ran 50 experiments where we gradually increased the number of RPC clients from 250 to 2000 with a number of RPC servers equals to 250, 500, 750 and 1000. One node was entirely dedicated to host the RabbitMQ message bus, while another one was fully dedicated to orchestrate all the experiments. Each of the 72 remaining nodes was thus hosting either multiple RPC clients or servers agents. Each of the clients sent a total of 10000 RPC requests, waiting 0.1 seconds between each request to avoid overloading the servers. The average latency and throughput of these requests are respectively plotted in Fig. 3 and Fig. 4 depending on the number of clients and servers.

Results clearly indicate that both the latency and throughput linearly increase with the number of clients, up to 1000 clients. With more than 1000 clients, the throughput reaches a limit, while the latency increases dramatically due to RabbitMQ limitations. Correlated with the internal queue size of the bus

and indirect measures like CPU and Memory we deduced that the bus was buffering a lot of messages when reaching this scale. It appeared, after further investigations that RPC servers blocked while handling requests leading to be considered as slow consumers from the bus perspective. In addition, we observed that the lower the number of RPC servers, the smaller the latency and the higher throughput. We suspect the bus delivery time to be dependent on the number of consumers a queue has, but it has to be confirmed in the near future.

These first experiments pave the way to larger scale benchmarks, deployed on different messaging bus technologies. Comparing these results to more distributed message bus for example is one of our close future work. Extending these experiments to other scenarios such as including network failures, or different communication patterns would provide developers a much better understanding of the message-oriented-middleware they aim to deploy.

B. Proposed Demonstration

We propose a live demonstration of the EnosStack. This demo will be based on the previous experimentation and will show how an experimenter can first validate its workflow in a local setup and then migrate it to an experimental platform (either Grid'5000 or Chameleon). We'll show how a list of parameters can be explored (each requiring a run of the workflow) while keeping all the wanted informations. Ultimately we'll give some hints on how the experimenter can create an environment allowing to analyse the results.

C. Reproducibility

As we put forward the importance of reproducible and repeatable experiments throughout this paper, we provide all the code and data that served to perform the experiments presented in section V. More specifically, the interested reader will find the followings:

- The EnosLib source code on GitHub [**?**]. This code is released under the GPL3.0
- The python code to deploy the experiments, as well as the default configuration files, also on GitHub [**?**] under GPLv3.0.
- All the docker images used to deploy services [**?**].
- The raw data results in JSON format, with all the parameters used [**?**].
- An online pre-written Jupyter notebook to extract meaningful results [**?**].
- The set of Gnuplot scripts and data files to plot the Figures in section V [**?**].

VI. Conclusion

In this paper, we presented the EnosStack, an open source software stack we specially designed to assist experimenters. Deploying real experiments that are repeatable and reproducible remains challenging, and we argued that experimenters should benefit from a LAMP-like stack to help them in running their experiments. We provided precise requirements for such a stack, and detailed how the abstractions we conceived fulfilled them. Finally we presented a real experiment deployed on Grid'5000 and showed how the EnosStack greatly eased its deployment.

This work has been supported by the Discovery Inria project lab and the joint lab between Orange and Inria (CRE amqp evaluation). Experiments presented in this paper were carried out using the Grid'5000 testbed, supported by a scientific interest group hosted by Inria and including CNRS, RENATER and several Universities as well as other organizations (see https://www.grid5000.fr).

References

[1] Docker images for the use case experiments. https://hub.docker.com/u/beyondtheclouds/.

[2] Python code of the enosstack use case experiments. https://github.com/msimonin/ombt-orchestrator.

[3] Repository of the enoslib. https://github.com/BeyondTheClouds/enoslib.

[4] Repository of the enosstack use case experiments. http://enos.irisa.fr/ombt-orchestrator/test_case_1_rabbitmq/.

[5] Shared jupyter notebook for the use case experiments. https://nbviewer.jupyter.org/url/enos.irisa.fr/ombt-orchestrator/test_case_1_rabbitmq/test_case_1.ipynb.

[6] Daniel Balouek, Alexandra Carpen Amarie, Ghislain Charrier, Frédéric Desprez, Emmanuel Jeannot, Emmanuel Jeanvoine, Adrien Lèbre, David Margery, Nicolas Niclausse, Lucas Nussbaum, Olivier Richard, Christian Pérez, Flavien Quesnel, Cyril Rohr, and Luc Sarzyniec. Adding virtualization capabilities to the Grid'5000 testbed. In Ivan I. Ivanov, Marten van Sinderen, Frank Leymann, and Tony Shan, editors, *Cloud Computing and Services Science*, volume 367 of *Communications in Computer and Information Science*, pages 3–20. Springer International Publishing, 2013.

[7] Adam Barker and Jano van Hemert. Scientific workflow: A survey and research directions. *Parallel Processing and Applied Mathematics, Springer Berlin Heidelberg*, 2008.

[8] Ronan-Alexandre Cherrueau, Dimitri Pertin, Anthony Simonet, Adrien Lèbre, and Matthieu Simonin. Toward a holistic framework for conducting scientific evaluations of openstack. In *Proceedings of the 17th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing, CCGRID 2017, Madrid, Spain, May 14-17, 2017*, pages 544–548, 2017.

[9] Fernando Chirigati, Rémi Rampin, Dennis Shasha, and Juliana Freire. Reprozip: Computational reproducibility with ease. *Proceedings of the International Conference on Management of Data (SIGMOD)*, 2016.

[10] Christian Collberg and Todd A. Proebsting. Repeatability in computer systems research. *Commun. ACM*, 2016.

[11] I. Jimenez, C. Maltzahn, A. Moody, K. Mohror, J. Lofstead, R. Arpaci-Dusseau, and A. Arpaci-Dusseau. The role of container technology in reproducible computer systems research. *IEEE International Conference on Cloud Engineering (IC2E)*, 2015.

[12] I. Jimenez, M. Sevilla, N. Watkins, C. Maltzahn, J. Lofstead, K. Mohror, A. Arpaci-Dusseau, and R. Arpaci-Dusseau. The popper convention: Making reproducible systems evaluation practical. *IEEE International Parallel and Distributed Processing Symposium Workshops*, 2017.

[13] Joe Mambretti, Jim Hao Chen, and Fei Yeh. Next generation clouds, the chameleon cloud testbed, and software defined networking (SDN). In *2015 International Conference on Cloud Computing Research and Innovation, ICCCRI 2015, Singapore, Singapore, October 26-27, 2015*, pages 73–79, 2015.

[14] Travis E. Oliphant. Python for scientific computing. *Computing in Science and Engg.*, 2007.

[15] Roger D. Peng. Reproducible research in computational science. *Science*, 2011.

[16] Thierry Rakotoarivelo, Maximilian Ott, Guillaume Jourjon, and Ivan Seskar. Omf: A control and management framework for networking testbeds. *SIGOPS Oper. Syst. Rev.*, 2010.

[17] Taylor J Sandve GK, Nekrutenko A and Hovig E. Ten simple rules for reproducible computational research. *PLoS Comput Biol*, 2013.